10 Rootkit Detection

I know not whether my native land
be a grazing ground for wild beasts or yet my home!
—Anonymous poet of Ma'arra

As we have shown throughout this book, rootkits can be difficult to detect, especially when they operate in the kernel. This is because a kernel rootkit can alter functions used by all software, including those needed by security software.

The same powers available to infection-prevention software are also available to a rootkit. Whatever avenues can be blocked to prevent rootkit intrusion can simply be unblocked. A rootkit can prevent detection or prevention software from running or working properly. In the end, it comes down to an arms race between the attacker and the defender, with a large advantage going to whichever one loads into the kernel and executes first.

That is not to say all is lost for the defender, but you should be aware what works today may not detect the rootkit of tomorrow. As rootkit developers learn what detection software is doing, better rootkits will evolve. The reverse is also true: Defenders will constantly update detection software as new rootkit techniques emerge.

In this chapter, we take a look at the two basic approaches to rootkit detection: detecting the rootkit itself, and detecting the behavior of a rootkit. Once you become familiar with these approaches, you will be in a better position to defend yourself.

Detecting Presence

Many techniques can be used to detect the presence of the rootkit. In the past, software such as Tripwire[1] looked for an image on the file system. This approach is still used by most anti-virus vendors, and can be applied to rootkit detection.

1. www.tripwire.org

The assumption behind such an approach is that a rootkit will use the file system. Obviously, this will not work if the rootkit runs only from memory or is located on a piece of hardware. In addition, if anti-rootkit programs are run on a live system that has already been infected, they may be defeated.[2] A rootkit that is hiding files by hooking system calls or by using a layered file filter driver will subvert this mode of detection.

Because software such as Tripwire has limitations, other methods of detecting rootkit presence have evolved. In the following sections, we will cover some of these methods, used to find a rootkit in memory or detect proof of the rootkit's presence.

Guarding the Doors

All software must "live" in memory somewhere. Thus, to discover a rootkit, you can look in memory.

This technique takes two forms. The first seeks to detect the rootkit as it loads into memory. This is a "guarding-the-doors" approach, detecting what comes into the computer (processes, device drivers, and so forth). A rootkit can use many different operating-system functions to load itself into memory. By watching these ingress points, detection software can sometimes spot the rootkit. However, there are many such points to watch; if the detection software misses any of the loading methods, all bets are off.

This was the problem with Pedestal Software's Integrity Protection Driver (IPD)[3]. IPD began by hooking kernel functions in the SSDT such as NtLoadDriver and NtOpenSection. One of your authors, Hoglund, found that one could load a module into kernel memory by calling ZwSetSystemInformation, which IPD was not filtering. After IPD was fixed to take this fact into account, in 2002, Crazylord published a paper that detailed using a symbolic link for \\DEVICE\\PHYSICALMEMORY to bypass IPD's protection.[4] IPD had to continually evolve to guard against the latest ways to bypass the protection software.

2. For best results, file integrity checking software should be run offline against a copy of the drive image.

3. It appears Pedestal (www.pedestalsoftware.com) no longer offers this product.

4. Crazylord, "Playing with Windows /dev/(k)mem," *Phrack* no. 59, Article 16 (28 June 2002), available at: www.phrack.org/phrack/59/p59-0x10.txt

The latest IPD version hooks these functions:

- ZwOpenKey
- ZwCreateKey
- ZwSetValueKey
- ZwCreateFile
- ZwOpenFile
- ZwOpenSection
- ZwCreateLinkObject
- ZwSetSystemInformation
- ZwOpenProcess

This seems like a long list of functions to watch! Indeed, the length of this list underscores the complexity of rootkit detection.

Moreover, the list is not complete. Yet another way to load a rootkit is to look for entry points into another process's address space. All the ways listed in Chapter 4, The Age-Old Art of Hooking, for loading a DLL into another process must also be watched. And all of this does not even cover every loading method discussed in this book.

Finding all the ways a rootkit might be loaded is just the first step in defending against rootkits. Load-detection techniques are belabored by the need to decide both what to guard and when to signal. For example, you can load a rootkit into memory using Registry keys. An obvious detection point would be to hook ZwOpenKey, ZwCreateKey, and ZwSetValueKey (as did IPD). However, if your detection software hooks these functions, how does it know which keys to guard?

Drivers are usually placed into the following key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services
```

This key is a good location to filter in your Registry-hook function, but a rootkit could also alter another key:

```
HKEY_LOCAL_MACHINE\System\ControlSet001\Services
```

This key can be used when the machine is booted into the previously known good configuration.

This example does not even begin to take into account all the Registry keys that deal with how application extensions are handled. And, consider that additional DLLs, such as Browser Helper Objects (BHOs), can be loaded into processes.

Detection software must also address the issue of symbolic links. Symbolic links are aliases for real names. A target you seek to protect could have more than one possible name. If your detection software hooks the system call table and a rootkit is using a symbolic link, the true target of the symbolic link will not have been resolved when your hook is called. Also, HKEY_LOCAL_MACHINE is not represented by that name in the kernel. Even if your detection software can hook all of these filter functions, the number of places to look seems infinite!

Still, let us assume you have discovered all the locations to watch in order to prevent rootkits from loading, and let's further assume you have resolved all the possible names of critical resources to protect. The difficulty you now face is in deciding when to signal. If you have detected a driver or a DLL loading, how do you know it is malware? Your detection software would need a signature for comparison, which assumes a known attack vector. Alternatively, your software could analyze the behavior of the module to try to determine whether it's malicious.

Both of these approaches are very hard to pursue successfully. Signatures require prior knowledge of the rootkit. This obviously doesn't work when a rootkit is yet unknown. Behavior detection is also difficult, plagued by false positives and false negatives.

Knowing when to signal is critical. This is an ongoing security battle, in which the anti-virus companies remain entrenched.

Scanning the "Rooms"

Scanning is the second technique for detecting rootkits in memory. In order to avoid the tedious labor of guarding all the entry points into the kernel or into a process's address space, you may want to scan memory periodically, looking for known modules or signatures of modules that correspond to rootkits. Again, this technique can find only known attackers. The advantage of this detection method is simplicity. The problem is that it doesn't prevent a rootkit from loading. In fact, it doesn't work unless the rootkit has already been loaded! If your software scans processes' address spaces, it will have to switch contexts into each process's address space, or do the virtual-to-physical address translation itself. If a kernel rootkit is already present, it can interfere with this memory walking.

Looking for Hooks

Another memory-based detection method is to look for hooks within the operating system and within processes. As we discussed in Chapters 4 and 5, there are many places where a hook can hide, including the following:

- Import Address Table (IAT)
- System Service Dispatch Table (SSDT), also known as the KeServiceDescriptorTable
- Interrupt Descriptor Table (IDT) with one per CPU
- Drivers' I/O Request Packet (IRP) handler
- Inline function hooks

When scanning for hooks, you suffer from all the shortcomings mentioned in the previous section on scanning the "rooms." The rootkit has already been loaded into memory and is executing; it may interfere with your detection methods. But one advantage to looking for hooks is that it's a generic approach. By looking for hooks, you do not have the problem of searching for known signatures or patterns.

The basic algorithm for identifying a hook is to look for branches that fall outside of an acceptable range. Such branches would be produced by instructions like `call` or `jmp`. Defining an acceptable range is not difficult (for the most part). In a process Import Address Table (IAT), the name of the module containing imported functions is listed. This module has a defined start address in memory, and a size. Those numbers are all you need to define an acceptable range.

Likewise for device drivers: All legitimate I/O Request Packet (IRP) handlers should exist within a given driver's address range, and all entries in the System Service Dispatch Table (SSDT) should be within the address range of the kernel process, ntoskrnl.exe.

Finding Interrupt Discriptor Table (IDT) hooks is a bit more difficult, because you do not know what the acceptable ranges should be for most of the interrupts. The one you know for sure, however, is the INT 2E handler. It should point to the kernel, ntoskrnl.exe.

Inline hooks are the hardest to detect, because they can be located anywhere within the function—requiring a complete disassembly of the function in order to find them—and because functions can call addresses outside the module's address range under normal circumstances. In the following sections, we will explain how to detect SSDT, IAT, and some inline hooks.

Getting the Address Ranges of Kernel Modules

To protect the SSDT or a driver's IRP handler table, you must first identify what an acceptable range is. To do this, you need a start address and a size. For kernel modules, you can call ZwQuerySystemInformation to find these.

You may be wondering whether this function cannot be hooked as well. It can, but if it is hooked and fails to return information for ntoskrnl.exe or

some driver you know is loaded, that is an indication that a rootkit is present.

To list all the kernel modules, you can call ZwQuerySystemInformation and specify that you are interested in the *class* of information called SystemModuleInformation. This will return a list of the loaded modules and each module's associated information. Here are the structures containing this information:

```
#define MAXIMUM_FILENAME_LENGTH 256

typedef struct _MODULE_INFO {
    DWORD d_Reserved1;
    DWORD d_Reserved2;
    PVOID p_Base;
    DWORD d_Size;
    DWORD d_Flags;
    WORD  w_Index;
    WORD  w_Rank;
    WORD  w_LoadCount;
    WORD  w_NameOffset;
    BYTE  a_bPath [MAXIMUM_FILENAME_LENGTH];
} MODULE_INFO, *PMODULE_INFO, **PPMODULE_INFO;

typedef struct _MODULE_LIST
{
    int         d_Modules;
    MODULE_INFO a_Modules [];
} MODULE_LIST, *PMODULE_LIST, **PPMODULE_LIST;
```

The GetListOfModules function will allocate the required memory for you, and return a pointer to this memory if it is able to get the system module information:

```
////////////////////////////////////////////////////////////////////
// PMODULE_LIST GetListOfModules
// Parameters:
//     IN PNTSTATUS pointer to NTSTATUS variable. This is useful for debugging.
// Returns:
//     OUT PMODULE_LIST     pointer to MODULE_LIST

PMODULE_LIST GetListOfModules(PNTSTATUS pns)
{
    ULONG ul_NeededSize;
    ULONG *pul_ModuleListAddress = NULL;
```

```
    NTSTATUS     ns;
    PMODULE_LIST pml = NULL;

    // Call it the first time to determine the size required
    // to store the information.
    ZwQuerySystemInformation(SystemModuleInformation,
                             &ul_NeededSize,
                             0,
                             &ul_NeededSize);
    pul_ModuleListAddress = (ULONG *) ExAllocatePool(PagedPool, ul_NeededSize);

    if (!pul_ModuleListAddress) // ExAllocatePool failed.
    {
        if (pns != NULL)

          *pns = STATUS_INSUFFICIENT_RESOURCES;

        return (PMODULE_LIST) pul_ModuleListAddress;
    }

    ns = ZwQuerySystemInformation(SystemModuleInformation,
                                  pul_ModuleListAddress,
                                  ul_NeededSize,
                                  0);
    if (ns != STATUS_SUCCESS)// ZwQuerySystemInformation failed.
    {
        // Free allocated paged kernel memory.
        ExFreePool((PVOID) pul_ModuleListAddress);
        if (pns != NULL)

           *pns = ns;
        return NULL;
    }
    pml = (PMODULE_LIST) pul_ModuleListAddress;

    if (pns != NULL)

       *pns = ns;
    return pml;
}
```

Now you have a list of all the kernel modules. For each of these, two important pieces of information were returned in the MODULE_INFO structure. One was the base address of the module, and the other was its size. You now have the acceptable range, so you can begin to look for hooks!

Finding SSDT Hooks

The following DriverEntry function calls the GetListOfModules function and then walks each entry, looking for the one named ntoskrnl.exe. When it is found, a global variable containing the beginning and end addresses of that module is initialized. This information will be used to look for addresses in the SSDT that are outside of ntoskrnl.exe's range.

```
typedef struct _NTOSKRNL {
     DWORD Base;
     DWORD End;
} NTOSKRNL, *PNTOSKRNL;

PMODULE_LIST   g_pml;
NTOSKRNL        g_ntoskrnl;

NTSTATUS DriverEntry(IN PDRIVER_OBJECT  DriverObject,
                     IN PUNICODE_STRING RegistryPath)
{
   int count;
   g_pml = NULL;
   g_ntoskrnl.Base = 0;
   g_ntoskrnl.End  = 0;
   g_pml = GetListOfModules();
   if (!g_pml)
      return STATUS_UNSUCCESSFUL;

   for (count = 0; count < g_pml->d_Modules; count++)
   {
      // Find the entry for ntoskrnl.exe.
      if (_stricmp("ntoskrnl.exe", g_pml->a_Modules[count].a_bPath + g_pml-
>a_Modules[count].w_NameOffset) == 0)
      {
         g_ntoskrnl.Base = (DWORD)g_pml->a_Modules[count].p_Base;
         g_ntoskrnl.End  = ((DWORD)g_pml->a_Modules[count].p_Base + g_pml-
>a_Modules[count].d_Size);
      }
   }
   ExFreePool(g_pml);

   if (g_ntoskrnl.Base != 0)
      return STATUS_SUCCESS;
   else
      return STATUS_UNSUCCESSFUL;
}
```

The following function will print a debug message if it finds an SSDT address out of acceptable range:

```
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase;
        unsigned int *ServiceCounterTableBase;
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} SDTEntry_t;
#pragma pack()

// Import KeServiceDescriptorTable from ntoskrnl.exe.
__declspec(dllimport) SDTEntry_t KeServiceDescriptorTable;

void IdentifySSDTHooks(void)
{
   int i;
   for (i = 0; i < KeServiceDescriptorTable.NumberOfServices; i++)
   {
      if ((KeServiceDescriptorTable.ServiceTableBase[i] <
                                     g_ntoskrnl.Base) ||
        (KeServiceDescriptorTable.ServiceTableBase[i] >
                                     g_ntoskrnl.End))

     {
        DbgPrint("System call %d is hooked at address %x!\n", i,
KeServiceDescriptorTable.ServiceTableBase[i]);
     }
   }
}
```

Finding SSDT hooks is very powerful, but do not be surprised if you find a few that are not rootkits. Remember, a lot of protection software today also hooks the kernel and various APIs.

In the next section, you will learn how to detect certain inline function hooks, which are discussed in Chapter 4.

Finding Inline Hooks

For simplicity in finding inline hooks, we will identify only detour patches that occur in the first several bytes of the function preamble. (A full-function disassembler in the kernel is beyond the scope of this book.) To detect these patches, we use the CheckNtoskrnlForOutsideJump function:

```
////////////////////////////////////////////////////
// DWORD CheckForOutsideJump
//
// Description:
//          This function takes the address of the function
//          to check. It then looks at the first few opcodes
//          looking for immediate jumps, etc.
//
DWORD CheckNtoskrnlForOutsideJump (DWORD dw_addr)
{
   BYTE  opcode = *((PBYTE)(dw_addr));
   DWORD hook   = 0;
   WORD  desc   = 0;

   // These are the opcodes for unconditional relative jumps.
   // Opcode 0xeb is a relative jump that takes one byte, so
   // at most it can jump 255 bytes from the current EIP.
   //
   // Currently not sure how to handle opcode 0xea. It looks
   // like jmp XXXX:XXXXXXXX. For now, I guess I will just
   // ignore the first two bytes. In the future, you should
   // add these two bytes as they represent the segment.
   if ((opcode == 0xe8) || (opcode == 0xe9))
   {
      // || (opcode == 0xeb) -> ignoring these short jumps
     hook |= *((PBYTE)(dw_addr+1)) << 0;
     hook |= *((PBYTE)(dw_addr+2)) << 8;
     hook |= *((PBYTE)(dw_addr+3)) << 16;
     hook |= *((PBYTE)(dw_addr+4)) << 24;
     hook += 5 + dw_addr;
   }

   else if (opcode == 0xea)
   {
      hook |= *((PBYTE)(dw_addr+1)) << 0;
      hook |= *((PBYTE)(dw_addr+2)) << 8;
      hook |= *((PBYTE)(dw_addr+3)) << 16;
      hook |= *((PBYTE)(dw_addr+4)) << 24;

      // Should update to reflect GDT entry,
      // but we are ignoring it for now.
      desc = *((WORD *)(dw_addr+5));
   }

   // Now that we have the target of the jump
   // we must check whether the hook is outside of
```

```
  // ntoskrnl. If it isn't, return 0.
  if (hook != 0)
  {
    if ((hook < g_ntoskrnl.Base) || (hook > g_ntoskrnl.End))
       hook = hook;
    else
       hook = 0;
  }

  return hook;
}
```

Given a function address in the SSDT, CheckNtoskrnlForOutsideJump goes to that function and looks for an immediate, unconditional jump. If one is found, it tries to resolve the address the CPU will jump to. The function then checks this address to determine whether it is outside the acceptable range for ntoskrnl.exe.

By substituting the appropriate range check, you can use this code to test for inline hooks in the first several bytes of any function.

Finding IRP Handler Hooks

You already have all the code necessary to find all the drivers in memory by using the GetModulesInformation function; and Chapter 4 covers how to locate the IRP handler table in a particular driver. To find driver IRP handler hooks, all you need to do is combine these two methods. You could even dereference each function pointer to search for inline function hooks within the handlers using the preceding code.

Finding IAT Hooks

IAT hooks are extremely popular with current Windows rootkits. IAT hooks are in the userland portion of a process, so they are easier to program than kernel rootkits, and do not require the same level of privilege. Because of this, you should make sure your detection software looks for IAT hooks.

Finding IAT hooks is very tedious, and implementing a search for them requires many of the techniques covered in previous chapters. However, those steps are relatively straightforward. First, change contexts into the process address space of the process you want to scan for hooks. In other words, your detection code must run within the process you are scanning. Some of the techniques for doing this are outlined in Chapter 4, in the Userland Hooks section.

Next, your code needs a list of all the DLLs the process has loaded. For the process, and every DLL within the process, your goal is to inspect the functions imported by scanning the IAT and looking for function addresses outside the range of the DLL the function is exported from. After you have the list of DLLs and the address range for each one, you can modify the code in the Hybrid Hooking Approach section of Chapter 4 to walk each IAT of each DLL to see whether there are any hooks. Particular attention should be paid to Kernel32.dll and NTDLL.DLL. These are common targets of rootkits, because these DLLs are the userland interface into the operating system.

If the IAT is not hooked, you should still look at the function itself to determine whether an inline hook is present. The code to do that is listed earlier in this chapter, in the CheckNtoskrnlForOutsideJump function; just change the range of the target DLL.

Once you are in a process's address space, there are several ways to find the list of process DLLs. For example, the Win32 API has a function called EnumProcessModules:

```
BOOL EnumProcessModules(
  HANDLE hProcess,
  HMODULE* lphModule,
  DWORD cb,
  LPDWORD lpcbNeeded
);
```

Pass a handle to the current process as the first parameter to EnumProcessModules, and it will return a listing of all the DLLs in the process. Alternatively, you could call this function from any process's address space. In that case, you would pass a handle to the target process you are scanning. The function, EnumProcesses, would then list all the processes. You do not have to worry whether there are hidden processes, because you do not care whether the rootkit has hooked its own hidden processes.

The second parameter to EnumProcessModules is a pointer to the buffer you must allocate in order to hold the list of DLL handles. The third parameter is the size of this buffer. If you have not allocated enough space to hold all the information, EnumProcessModules will return the size needed to store all the DLL handles.

With a handle to every DLL in the process returned by EnumProcessModules, you can get each DLL's name by calling the GetModuleFileNameEx function. Another function, GetModuleInformation, returns the

DLL base address and size for each DLL handle you use as the second parameter. This information is returned in the form of a MODULE_INFORMATION structure:

```
typedef struct _MODULEINFO {
  LPVOID lpBaseOfDll;
  DWORD SizeOfImage;
  LPVOID EntryPoint;
} MODULEINFO, *LPMODULEINFO;
```

With the name of the DLL, its start address, and its length, you have all the data necessary to determine an acceptable range for the functions it contains. This information should be stored in a linked list so that you can access it later.

Now begin to walk each file in memory, parsing the IAT of each DLL just as illustrated in the Hybrid Hooking Approach section in Chapter 4. (Remember that each process and each DLL's IAT can hold imports from multiple other DLLs.) This time, though, when you parse a process or a DLL looking for its IAT, identify each DLL it is importing. You can use the name of the DLL being imported to find the DLL in the stored linked list of DLLs. Now compare each address in the IAT to its corresponding DLL module information.

The preceding technique requires the EnumProcesses, EnumProcessModules, GetModuleFileNameEx, and the GetModuleInformation APIs. The attacker's rootkit could have hooked these calls. If you want to find the list of DLLs loaded in a process without making any API calls, you can parse the Process Environment Block (PEB). It contains a linked list of all the loaded modules. This technique has long been used by all sorts of attackers, including virus writers. In order to implement this technique, you will have to write a little Assembly language. The Last Stage of Delirium Research Group has written a very good paper[5] that details how to find the linked list of DLLs within a process.

> **Rootkit.com**
> The previously shown sections of code for finding IAT, SSDT, IRP, and Inline hooks are implemented in the tool VICE, available at:
> www.rootkit.com/vault/fuzen_op/vice.zip

5. The Last Stage of Delirium Research Group, "Win32 Assembly Components" (updated 12 December 2002), available at: http://lsd-pl.net/windows_components.html

Tracing Execution

Another way to find hooks in APIs and in system services is to trace the execution of the calls. This method was used by Joanna Rutkowska in her tool Patchfinder 2.[6] The premise is that hooks cause extra instructions to be executed that would not be called by unhooked functions. Her software baselines several functions at boot, and requires that at that time the system is not hooked. Once this baseline is recorded, the software can then periodically call the functions again, checking to see whether additional instructions have been executed in subsequent calls when compared to the baseline.

Although this technique works, it suffers from the fact that it requires a clean baseline. Also, the number of instructions a particular function executes can vary from one call to the next, even if it is not hooked. This is largely due to the fact that the number of instructions depends on the data set the function is parsing. What is an acceptable variance is a matter of opinion. Although Rutkowska does state that, in her tests, the difference between a hooked function and an unhooked function was significant when tested against known rootkits, that difference could depend upon the sophistication of the attacker.

Detecting Behavior

Detecting behavior is a promising new area in rootkit detection. It is perhaps the most powerful. The goal of this technique is to catch the operating system in a "lie." If you find an API that returns values you know to be false, not only have you identified the presence of a rootkit, but you have also identified what the rootkit is trying to hide. The behavior you are looking for is the lie. A caveat to this is that you must be able to determine what the "truth" is without relying upon the API you are checking.

Detecting Hidden Files and Registry Keys

Mark Russinovich and Bryce Cogswell have released a tool called Rootkit-Revealer.[7] It can detect hidden Registry entries as well as hidden files. To

---

6. J. Rutkowska, "Detecting Windows Server Compromises with Patchfinder 2" (January 2004), available at: www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf

7. B. Cogswell and M. Russinovich, *RootkitRevealer,* available at: www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml

determine what the "truth" is, RootkitRevealer parses the files that correspond to the different Registry hives without the aide of the standard Win32 API calls, such as RegOpenKeyEx and RegQueryValueEx. It also parses the file system at a very low level, avoiding the typical API calls. RootkitRevealer then calls the highest level APIs to compare the result with what it knows to be true. If a discrepancy is found, the behavior of the rootkit (and, hence, what it is hiding) is identified. This technique is fairly straightforward, yet very powerful.

Detecting Hidden Processes

Hidden processes and files are some of the most common threats you will face. A hidden process is particularly threatening because it represents code running on your system that you are completely unaware of. In this section, you will learn different ways to detect processes the attacker does not want you to see.

Hooking SwapContext

Hooking functions is useful during detection. The SwapContext function in ntoskrnl.exe is called to swap the currently running thread's context with the thread's context that is resuming execution. When SwapContext has been called, the value contained in the EDI register is a pointer to the next thread to be swapped in, and the value contained in the ESI register is a pointer to the current thread, which is about to be swapped out. For this detection method, replace the preamble of SwapContext with a five-byte unconditional jump to your detour function. Your detour function should verify that the KTHREAD of the thread to be swapped in (referenced by the EDI register) points to an EPROCESS block that is appropriately linked to the doubly linked list of EPROCESS blocks. With this information, you can find a process that was hidden using the DKOM tricks outlined in Chapter 7, Direct Kernel Object Manipulation. The reason this works is that scheduling in the kernel is done on a thread basis, as you will recall, and all threads are linked to their parent processes. This detection technique was first documented by James Butler et. al.[8]

Alternatively, you could use this method to detect processes hidden by hooking. By hooking SwapContext, you get the true list of processes. You

8. J. Butler et al., "Hidden Processes: The Implication for Intrusion Detection," *Proceedings of the IEEE Workshop on Information Assurance* (United States Military Academy, West Point, NY), June 2003.

can then compare this data with that returned by the APIs used to list processes, such as the NtQuerySystemInformation function that was hooked in the section Hooking the System Service Descriptor Table in Chapter 4.

Different Sources of Process Listings

There are ways to list the processes on the system other than going through the ZwQuerySystemInformation function. DKOM and hooking tricks will fool this API. However, a simple alternative like listing the ports with netstat.exe may reveal a hidden process, because it has a handle to a port open. We discuss using netstat.exe in Chapter 4.

The process CSRSS.EXE is another source for finding almost all the processes on the system. It has a handle to every process except these four:

- The Idle process
- The System process
- SMSS.EXE
- CSRSS.EXE

By walking the handles in CSRSS.EXE and identifying the processes to which they refer, you obtain a data set to compare against the list of processes returned by the APIs. Table 10–1 contains the offsets necessary in order to find the handle table of CSRSS.EXE. Within the EPROCESS block of every process is a pointer to a structure that is its HANDLE_TABLE. The HANDLE_TABLE structure contains a pointer to the actual handle table, among other information. For further information on how to parse the handle table, see Russinovich and Solomon's book, *Microsoft Windows Internals*.[9]

Table 10–1 Offsets for finding handles from an EPROCESS block.

	Windows 2000	Windows XP	Windows 2003
Offset to Handle Table in EPROCESS	0x128	0xc4	0xc4
Offset to the actual table within the Handle Table Structure	0x8	0x0	0x0

9. M. Russinovich and D. Solomon, *Microsoft Windows Internals, Fourth Edition* (Redmond, Wash.: Microsoft Press, 2005), pp. 124–49.

Another technique exists for identifying the list of processes without calling a potentially corrupted API. You know from our earlier discussion that every process's EPROCESS block has a pointer to its handle table. It turns out that all these handle table structures are linked by a LIST_ENTRY, similarly to the way all processes are linked by a LIST_ENTRY (see Chapter 7). By finding the handle table for any process and then walking the list of handle tables, you can identify every process on the system. As of this writing, we believe this is the technique used by BlackLight[10] from the antivirus company F-Secure.

In order to walk the list of handle tables, you need the offset of the LIST_ENTRY within the handle table structure (in addition to the offset within the EPROCESS block of the pointer to the handle table, which you have from the Table 10–1). The HANDLE_TABLE structure also contains the PID of the process that owns the handle table. The PID is also found at different offsets depending on the version of the Windows operating system. The offsets to identify every process based upon its PID are given in Table 10–2.

As you traverse each process using the LIST_ENTRY values, you can find the owning PIDs. Now you have another data set to compare against if the Win32 API fails to list a particular process. The following function lists all the processes on the system by walking the linked list of handle tables:

```
void ListProcessesByHandleTable(void)
{
   PEPROCESS eproc;
   PLIST_ENTRY start_plist, plist_hTable = NULL;
   PDWORD d_pid;
   // Get the current EPROCESS block.
```

Table 10–2 Offsets used to walk the handle tables and ID the processes.

	Windows 2000	Windows XP	Windows 2003
Offset to LIST_ENTRY within Handle Table	0x54	0x1c	0x1c
Offset to Process ID within Handle Table	0x10	0x08	0x08

10. *F-Secure BlackLight* (Helsinki, Finland: F-Secure Corporation, 2005): www.f-secure.com/blacklight/

```
   eproc = PsGetCurrentProcess();
   plist_hTable = (PLIST_ENTRY)((*(PDWORD)((DWORD) eproc +
                  HANDLETABLEOFFSET)) + HANDLELISTOFFSET);
   start_plist = plist_hTable;
   do
   {
      d_pid = (PDWORD)(((DWORD)plist_hTable + EPROCPIDOFFSET)
              - HANDLELISTOFFSET);
      // Print the Process ID as a debug message.
      // You could store it to compare to API calls.
      DbgPrint("Process ID: %d\n", *d_pid);
      // Advance.
      plist_hTable = plist_hTable->Flink;
   }while (start_plist != plist_hTable);
}
```

This is just another way to identify a hidden process, but it is very effective. If the rootkit does not alter this list in the kernel, which can be difficult to do, your detection method will catch its hidden processes. There are other, similar structures in the kernel that could be used in this way as well. Detection techniques are evolving as fast as rootkits are.

Conclusion

This chapter has shown you many different ways to detect rootkits. We have covered practical implementations, and discussed the theory behind other techniques.

Most of the methods in this chapter have focused on detecting hooks and hidden processes. Whole books could be written on file-system detection, or on detecting covert communication channels. By identifying hooks, though, you will be well on your way to detecting most public rootkits.

No detection algorithm is complete or foolproof. The art of detection is just that—an art. As the attacker advances, the detection methods will evolve.

One drawback of spelling out both rootkit and detection methodologies is that this discussion favors the attacker. As methods to detect an attacker are explained, the attacker will alter her methodology. However, the mere fact that a particular subversion technique has not been written up in a book or presented at a conference does not make anyone any safer. The level of sophistication in the attacks presented in this book is beyond the reach of the majority of so-called "hackers," who are basically script-kiddies. We

hope the techniques discussed in this publication will become the first methods that security companies and operating system creators begin to defend against.

More-advanced rootkit techniques and their detection are being developed as you read these words. Currently, we are aware of several efforts to cloak rootkits in memory so that even memory scanning is corrupted. Other groups are moving to hardware with embedded processors in order to scan kernel memory without relying upon the operating system.[11] Obviously these two groups will be at odds. Since neither implementation is available for public scrutiny, it is hard to say which one has the upper hand. We are sure that each one will have its own limitations and weaknesses.

The rootkits and detection software mentioned in the previous paragraph represent the extremes. Before you begin to worry about these new tools, you need to address the most common threats. This book has shown you what they are, and where the attacker is likely to go.

Recently we have seen companies showing their first signs of interest in rootkit detection. We hope this trend will continue. Having more-informed consumers will cause protection software to advance. The same can be said for having more-informed attackers.

As we stated in Chapter 1, corporations are not motivated to protect against a potential attack until there is an attack. You are now that motivation!

---

11. N. Petroni, J. Molina, T. Fraser, and W. Arbaugh (University of Maryland, College Park, Md.), "Copilot: A Coprocessor Based Kernel Runtime Integrity Monitor," paper presented at Usenix Security Symposium 2004, available at: www.usenix.org/events/sec04/tech/petroni.html